INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

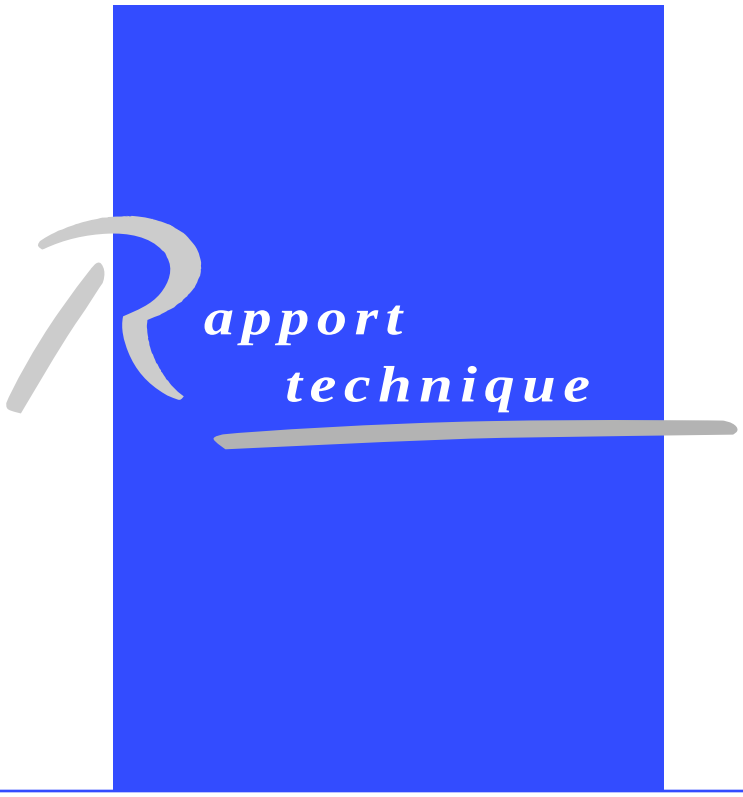
AS, for Abstract Syntax Manual - V1.0

Thierry Despeyroux

N° 0197

Septembre 1996

THÈME 2



ISSN 0249-0803

AS, for Abstract Syntax Manual - V1.0

Thierry Despeyroux*

Thème 2 — Génie logiciel
et calcul symbolique
Projet CROAP

Rapport technique n°0197 — Septembre 1996 — 36 pages

Abstract: This is the manual for AS, an abstract syntax specification formalism. The main features of this formalism are modularity and support for second-order abstract syntaxes. AS is the first formalism from the CLF (Computer Languages Factory), a forthcoming set of tools and specification formalisms for quick prototyping and complete implementation of computer languages syntaxes and semantics. This version of AS may be used under the Centaur system for first-order features only. The second-order features will be useful only when a higher-order version of Typol will be distributed.

Key-words: abstract syntax, programming environment, Centaur, software engineering

(Résumé : tsvp)

* Thierry.Despeyroux@sophia.inria.fr http://www.inria.fr/croap/personnel/Thierry.Despeyroux.html

Unité de recherche INRIA Sophia-Antipolis
2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex (France)
Téléphone : (33) 93 65 77 77 – Télécopie : (33) 93 65 77 65

AS, pour Abstract Syntax
Manuel - V1.0

Résumé : Ce document est le manuel de AS, un langage de spécification de syntaxes abstraites. Les principaux atouts de ce langage sont sa modularité et la prise en compte des syntaxes abstraites de second ordre. AS est le premier élément d'un ensemble d'outils et formalismes de spécification (CLF, pour Computer Languages Factory) dont le but est de permettre le prototypage rapide et l'implémentation complète de la syntaxe et de la sémantique de langages informatiques. Cette version de AS est disponible pour le système Centaur uniquement dans le cas premier ordre. Le second ordre ne sera utile que quand une version d'ordre supérieur de Typol sera disponible.

Mots-clé : syntaxe abstraite, environnement de programmation, Centaur, génie logiciel

1 Introduction

The specification language AS is a successor to a subpart of Metal [7], the abstract and concrete syntax specification language created for Mentor [5] and now used in Centaur [1].

Even if this new formalism has been developed under Centaur, and if the definitions written using AS are compatible with most of the rest of the Centaur system (at least for the first-order specifications), the aim of this formalism is to be a specification formalism, independent from a particular implementation or host system. This means that if it will be possible to compile AS specifications for the Centaur system or an eventual successor, we want also to be able to produce independent "batch" implementations. AS is the first language from a forthcoming family of specification languages (the CLF, for Computer Languages Factory) that wants to cover also concrete syntax and semantics, enhancing the trilogy Metal/PPML/Typol, and designed for quick prototyping and complete implementation of computer languages.

AS is a successor for a subpart of Metal only, more precisely the abstract syntax part of Metal. To produce complete programming environments or compilers using AS one will have to wait for the rest of the family, more precisely to the concrete syntax specification formalism. But it is possible to use it now as soon as one does not need a parser for a particular formalism. It is possible to use PPML and Typol for a formalism defined with AS. AS may be in particular used now to define "environments" used in Typol specifications as well as a target language in a translator or a code generator specified in Typol. AS as been already used to define several languages for the Centaur system: a subset of prolog used by its own code generator, the environment used by its own type-checker, and also some "real" languages such as C and Java.

AS is also an extension to Metal. It allows the definition of modular and second-order abstract syntaxes. If modularity can be used right now, second-order features will be useful only when a higher-order version of Typol will be available.

The next section is a rational that compares AS with other specification formalisms and explains the choices that have been made in the design of the language. Follow a tutorial, the user's manual that explains how to manipulate, compile and use the result of the compilation of AS specifications, and the reference manual that describes all the features of the language. The section about errors explains how to understand warnings and error messages issued by the type-checker or by the compiler. The last section contains implementation notes that will be useful only for implementors or advanced users.

This manual as been designed to be a traditional manual and to be accessible as online documentation. To achieve this goal, both LaTeXand LaTeX2HTML has been used to produce an hypertext document. This document is directly accessible from the Centaur environment for AS. In particular error messages produced by the type-checker can refer directly to the corresponding documentation using a Web browser running in a server mode as Netscape. The printed version of the manual contains also some links to help the user to navigate in the document. These links are indicated by the sign "▷".

2 Rational

This section can be skipped by a user that only wants to use AS. However, it will help to understand the reasons why the AS formalism has been created and the choices that have been made in its design.

2.1 The Computer Languages Factory

Abstract syntax is now a traditional way of defining and representing the syntax of programming languages. It tends to give the deep structure of programs, that is in general different from the parse tree. First-order abstract syntax is commonly used in programming tools such as syntactical editors, programming environments or compilers.

To build some generic tools that manipulates programs, one must find a way to give a specification of the language in which these programs are written. This may be a table as is was the case at the beginning of the table driven syntactical editor Mentor [5], a set of primitives to create types and constructors as in the VTP (Virtual Tree Processor) [9] used by Centaur [1], or a specialized formalism that will be compiled in some way to a lower level specification as it is done by Metal [5] that was designed for Mentor and reused for Centaur, and for SDF [8].

Metal has been used with success for many years. Of course, the definition formalisms used in Centaur (Metal itself, PPML, Typol) are defined with Metal, but it has also been used for a large variety of programming languages: Pascal, ADA, Esterel, VHDL, Java, and even a commercial application of Centaur for Fortran.

However, after using Centaur for some years and listening to users it appears that some great enhancements may be achieved by redesigning the specification formalisms of Centaur. First of all, there is no complete type-checker for Metal, and that leads of course to problems that arise very late when specifying a language for Centaur, in general when using Typol to express the semantic properties of the language. Second, each formalism of the trilogy Metal/PPML/Typol has its own idea of variables and schemes. Obviously, schemes should be a sub-language common to the three formalisms. Third, type inclusion is not explicit in Metal, and that may create some problems as the intersection of two types may be unnamed. Finally, even if it is possible to run Centaur in batch mode, one wants to get some lighter implementations independent from Lisp, the implementation language of Centaur.

Some other remarks ask to rebuild the specification languages of Centaur. First, there is no modularity in Metal, and because this lack of modularity there is no abstract syntax modularity in PPML, neither in Typol. Second, there is no manner to express bindings and this means that for languages based on λ-calculi, for example, one will have to give an implementation of substitutions to express the semantic properties of these languages. Last, run-time performance is no more a problem now, or may be less important that it has been in the past. What we need is a way to perform, with a reasonable degree of safety, some quick language prototyping, taking into account semantical problems. Maintaining and reuse of specifications must also be easy.

The Computer Languages Factory (CLF) is a set of tools and specification formalisms together with their implementations. It will be design for quick prototyping and complete implementation of computer languages, enhancing the trilogy Metal/PPML/Typol.

The following sections describe with more details some features of the first member of the CLF: AS.

2.2 Abstract Syntax

An abstract syntax is a type system designed to give a natural representation of programs as typed trees. AS follows what was already done by Metal and SDF [8] (even if in SDF the abstract syntax is generated from a concrete syntax specification). This means that an abstract syntax is a set of basic types (also called phyla) with explicit type inclusion, together with a set of constructors (also called operators) that are defined by their signatures.

▷ *Starting with an Example* 3.1 *p.*8

Notice that this presentation of an abstract syntax, even if it was used at the very beginning of the Mentor project [4], was not directly used in Metal for some historical implementation reasons. In Metal, operators are defined only by the type of their sons, and phyla are defined as set of operators. This was a consequence of the table-driven design of Mentor, rather than a specification driven view. A table can be viewed as a way of encoding an abstract syntax [3].

2.3 Isolating Abstract Syntax from Concrete Syntax

For many years (and even now), a computer language was mainly defined by its concrete syntax, and this concrete syntax was supposed to be unique. This is the reason why in Metal the abstract syntax and the concrete syntax of a computer language was given in a unique specification. However, experiments with the Centaur system show that it is very often useful to have several concrete representations for the same program. In Centaur this may be achieved by specifying several pretty-printers for the same language. But until now, there is still a unique parser for each formalism.

There is no good reason for this asymmetry, but to be able to specify multiple parsers for one formalism, one has to separate the definition of the abstract syntax, that must be unique, from the definitions of the concrete syntaxes.

This option may appear opposite to what is done in SDF [8] or Syn [2]. One of the goals of SDF+ASF is to give the possibility to express semantical properties of a computer language using its concrete syntax. This is why it seems reasonable to generate the abstract syntax from the concrete one in this context. When more than one concrete syntax exists for a unique computer language, this method of hiding the abstract syntax to the user cannot be used anymore. It should be possible to see the abstract syntax definition (even if it has been generated by some mean) and to reuse it for an alternate concrete syntax.

2.4 Modularity

Modularity in computer languages exists more often that one may imagine. It is the case for example in Lex or Yacc in which part of the rules may be C code. In the trilogy Metal/PPML/Typol, all the three formalisms contains a subpart for expressing schemes. This subpart should be isolated and shared between the three formalisms (not only the syntax, but also the type-checker for example).

In these cases, the domain of the constructor of one language may contain a phylum belonging to another language. The type system of the imported formalism is simply reused (or imported), and not modified. This sort of modularity, very closed to type inclusion, may be called formalism inclusion.

▷ *Formalism Inclusion* 3.6 *p.*12

An other kind of modularity is related to dialects of a computer language. Each Pascal or Fortran compiler accepts its own special features. Thus some programs are portable because they use features implemented in all the available compilers (we may say that these programs belong to the standard of the language). Some other programs can be compiled only using some compilers because they use some extensions of the standard.

It should be possible to define the common part (the standard part) of different dialects, then to define each dialect as an extension of the standard. In this case, the types (phyla) from the standard are modified by adding some new constructors (the extensions), and there is an injection sending standard programs into non-standard ones. Again, the semantics of a dialect should reuse the semantics of the standard. This sort of modularity may be called formalism extension.

▷ *Formalism Extension* 3.7 *p.*13

One may also think of combining these two principles of abstract syntax modularity.

2.5 Second-Order Abstract Syntax

One of the problems of computer languages comes from the fact that one (a human or a compiler) has to connect the uses of identifiers with their definitions. This problem is known as the binding problem, with a myriad of variants or related topics: scope of declarations, visibility of a variable, nesting of environments, visibility holes, capture of variables, substitutions rules, etc.

The need to incorporate the notion of binding into abstract syntaxes leads to what is called higher-order abstract syntax. Higher-order abstract syntax has already been used in some systems as λ-Prolog [10, 11] or Elf [12] and some experiments of defining or implementing the semantics of computer formalisms has been made [6]. However, no system has integrated this notion to produce complete programming environments. AS will integrate the notion of higher-order abstract syntax, restricted to the second-order case that seems sufficient for most of practical applications.

Note that if some languages (derived for example from the simply typed λ-calculus) have an obvious binding rule (let's say that bindings can be decided at parse time), this is not the case for some other ones in which bindings can only be made after type-checking. This

is the case for example for Pascal (due to the "with" construction), or for ADA (due to overloading). For this reason, AS will provide a way to define at the same time a first-order an a second-order abstract syntax for the same formalism.

▷ *Binders* 3.5 *p.*11

3 Tutorial

A natural and common way of representing programs inside a computer is to use trees (encoded as terms for example in Prolog, of structures or records in some other languages). Obviously these trees are not ordinary trees, but there is a type structure that tells which trees are well formed (for example the definition of a particular language may state that a statement cannot be an argument of an arithmetic expression). This type structure is called abstract syntax.

An abstract syntax consists of mutually recursively defined types, that we call phyla, a cartesian product of phyla, and typed constructors called operators. We have chosen to admit in this scheme type inclusions (that are not present in every abstract syntax systems). This means that even if an operator is defined to belong to a unique phylum, it can also be assigned many types due to type inclusions, expressing the fact that it can appear in different contexts (for example, an identifier can be the first argument of an assignment, but can also appear in an expression). We have also chosen to admit homogeneous lists in our type schemes.

Operators are defined by their signatures (the phylum to which it belongs and the number and types of its descendents, or sons). Up to now, this type structure is the one used in Metal and SDF, bringing to what is known as first-order abstract syntax. The formalism AS also admit some functional types to express syntactical bindings, bringing to second-order abstract syntax.

Choosing an abstract syntax for a particular language is not always an obvious task. It is a question of taste and experience. A good rule is to try to reflect the concepts and the semantics of the language. But it is sometimes easier if the language already exists to follow the concrete syntax if it is reasonable. One have also to make a compromise between a very strict type system that may need a large number of phyla and operators and a more permissive but more compact system (in this case the missing constraints will have to be checked later on in the semantics, for example by a type-checker). There is also the temptation to sacrifice the abstract syntax to some practical and independent reasons, for example when we want to generate a syntactical editor or a pretty printer. These are bad reasons that reveal some lacks of the tools that are used, and should not be taken into account.

This section give some examples that form a good basis to construct abstract syntaxes.

3.1 Starting with an Example

To get a global view of AS, let us take a small example that will not look too strange to old Centaur users: the EXP example.

```
abstract syntax of EXP is

assign : Var # Exp -> Exp;
plus, minus, prod : Exp # Exp -> Exp;
```

```
uminus : Exp -> Exp;

#integer : integer -> Int;
variable : string -> Var;

Var < Exp;
Int < Exp;

exp_s : EXP+ -> Exp_s;

Atoms = Var + Int;

end Exp;
```

This definition defines tree kinds of objects: one formalism (`EXP`), five phyla (`Exp, Int, Var, Exp_s and Atoms`), eight operators (`assign, plus, minus, prod, uminus, #integer, variable and exp_s`). As `integer` is a keyword of AS, the name `integer` has been used for an operator by prefixing it with a sharp symbol.

▷ *Keywords* 5.1.1 *p.*17

Only three sorts of statements are used: operators definitions (giving their signatures), types inclusions, and types unions. In signatures, the sharp symbol (`#`) represents the product of phyla and the notation "`P+`" reads as "list of `P`". Type inclusion is represented by "`<`" and type union by "`+`".

▷ *Definition of Operators* 5.4 *p.*19 - *Definition of Phyla* 5.5 *p.*21 - *Lists* 3.4 *p.*11

The types `integer` and `string` are predefined.

▷ *Predefined Types* 5.6 *p.*22

3.2 Expressions

The first thing to notice about expressions is that the notion of precedence does not appear in the abstract syntax. The tree form of the structure is never ambiguous. It is the role of the parser to take into account the precedence rules and to create the correct abstract syntax tree. Parenthesis (and in particular redundant parenthesis) do not appear in an abstract syntax tree. It is the role of an un-parser to add necessary (and only those) parenthesis when producing a textual form.

A first idea to define expressions is to create one operator for each operation.

```
plus, minus : Exp # Exp -> Exp;
```

If the number of similar operators is important, one may prefer to share the notion of binary expression.

```
bexp : Exp # Bop # Exp -> Exp;
plus, minus : -> Bop;
```

Notice here two "constant" operators (they was called singletons in Metal): `plus` and `minus`.

The two methods may also be mixed. For example, access to an array is also a binary expression, but one prefers to distinguish this operator from arithmetic operations.

```
bexp : Exp # Bop # Exp -> Exp;
plus, minus : -> Bop;
index : Exp # Exp -> Exp;
```

▷ *Definition of Operators* 5.4 *p.*19

3.3 Statements

Statements form in general the simple part of an abstract syntax. Again (as for precedence in expressions), problems such as "dangling else" do not belong to the abstract syntax word. And again some choices must be made. This is the case for optional parts. The typical case is those of an if statement in which the else part is optional. A first attempt can be:

```
ifthen : Exp #  Statement -> Statement;
ifthenelse: Exp # Statement # Statement -> Statement;
```

One can want to have a unique operator to unify the two cases. But this means that (if we have to write the semantics after that) we can give a semantics to the null statement.

```
if : Exp # Statement # Opt_Statement -> Statement;
null_statement : -> Null_Statement;
Opt_Statement = Statement + Null_Statement;
```

The more natural solution uses lists of statements, with the possibility to have an empty list of statements.

```
if : Exp # Statement_s # Statement_s -> Statement;
statement_s : Statement* -> Statement;
```

The notations "`P*`" and "`P+`" where `P` is the name of a phyla read as "list of `P`". In the first case the list can be empty, while the second case indicates a non empty list.

In this example, the "then" part of the if-statement can be an empty list. Again, one may have to make a choice between a strict abstract syntax introducing a phylum for non-empty lists of statements, and a more permissive one.

▷ *Lists* 3.4 *p.*11 - *Definition of Operators* 5.4 *p.*19

3.4 Lists

The need for homogeneous lists came naturally when designing an abstract syntax. However one may have to make a choice between "+-lists" (which cannot be empty) and "*-lists" (which can be empty). This is in particular the case when both kind of lists (of the same base elements) can be found at different points in a program.

The easiest solution is to have only a "*-list" in the abstract syntax. Allowing at some point empty lists that must be rejected later on, for example by a type-checker.

```
exp_s : Exp* -> Exp_s;
```

A more rigorous solution is to use a "+-list", and a special node allowed when the list is optional.

```
exp_s : Exp+ -> Exp_s;
none : -> Opt_Exp;
Exp < Opt_Exp;
```

The same sort of problem occurs when we have to give an abstract syntax to bounded lists, or even lists.

▷ *List operators* 5.4.3 *p.*20

3.5 Binders

The typical case of binder comes from the simple λ-calculus (or ML-like languages). If we use a first-order abstract syntax we can simply define the expression $\lambda x.E$ as:

```
lambda : Id # Exp -> Exp;
```

If we choose a second-order abstract syntax, as the syntactical type of $\lambda x.E$ is `Exp -> Exp` we declare:

```
lambda : (Id:Exp -> Exp) -> Exp;
```

This definition express both the fact that a `lambda` has a functional type and that the binder is concretely represented by an identifier.

Note that the first-order abstract syntax generated by this two declarations are identical.

The abstract syntax for a binder may seems distant from the concrete syntax. For example, in the typed λ-calculus expression $\lambda x : T.E$, T is not in the scope of x, and so the correct abstract syntax (even in the first-order case) for this expression is:

```
lambda : Type # Binder -> Exp;
binder : (Id:Exp -> Exp) -> Binder;
```

AS does not allow a general higher-order abstract syntax, as for example λ-prolog. Only a second-order abstract syntax is permitted, and an operator must be declared for each higher-order type. This means that declarations like `op : (X -> X) -> (X -> X) -> X` in λ-Prolog are not possible in AS. One will have to give more operators as in the following example.

```
op : Binder # Binder -> X;
binder : (Id:X -> X) -> Binder;
```

▷ *Second-order operators* 5.4.4 *p.*20

3.6 Formalism Inclusion

Typol users will found in formalism inclusion an easy way to design "environments" without polluting the abstract syntax of their object language, nor duplicating some parts of its abstract syntax.

Suppose that we have defined a formalism containing some identifiers and integers:

```
abstract syntax of L is
...
id : string -> Id;
int : integer -> Int;
...
end L;
```

When writing a dynamic semantics for this language, we may want to describe the memory as a pair of identifiers and integer values, where these identifiers and values belong to L:

```
abstract syntax of L_env is

memory : Pair * -> Mem;
pair : Id::L # Int::L -> Pair;

end L_env;
```

The notation "`P::L`" indicates that `P` is a phylum belonging to the formalism `L`.

Formalism inclusion can also be used together with type inclusion:

```
Exp::L < Expression;
```

Formalism inclusion can be used to define languages which share some parts of their syntax with an other formalism. This is the case for example for Lex and Yacc in which the action part of a rule is a C statement.

▷ *Typed Names* 5.3 *p.*18

3.7 Formalism Extension

Suppose that we have already defined an abstract syntax and possibly some semantics for a computer language. Even if a standard have been accepted for this language, it will certainly be the case that some compilers accept some proprietary features, for example "tag" and "exit" statements to provide easy error recovery or exceptions. To take this extension into account, we want to inherit most of what have been done for the standard version of the language.

```
abstract syntax of L_Best_Inc
  extends L with

tag, exit : Id # Exp -> Statement;

end L_Best_Inc;
```

This definition will create a new formalism tailored for the Best Inc. compiler. This formalism is different from the initial one. A procedure will be provided to coerce L programs into L_Best_inc ones, and L_Best_inc programs that do not make use of the tag and exit statements into L programs.

AS does not implement formalism restriction. However this can be easily simulated with formalism extension.

▷ *Extension of a Formalism* 5.8 *p.*22

4 User's Manual

To define a new abstract syntax one have to write a specification, then type-check this specification and compile it.

In the present implementation, two compilers exist. One produces some Lisp code used to create a Centaur formalism, the second produces some Prolog code needed to compile or execute Typol specifications, or to compile an abstract syntax that include the present one. Centaur users will have to use both of them.

The source file can be produced with any editor. However, starting to enjoy Centaur syntactical editing with a simple language such as AS is not a bad idea.

Before complete bootstrapping of the CLF, type-checking and compiling of AS specification can only be done within Centaur.

4.1 Files Naming Convention

The AS specification of a formalism named `L` must be founded in a file named `L.as`. Be cautious that the name of the formalism and the name used for the source file are identical (the Centaur environment for AS will warn you if it is not the case). This naming convention is used by Centaur and Typol for automatic file loading.

After successful type-checking and compiling, three files are produced. The lisp code will be found in `L.as.ll` and the prolog code will be found in `L.as.sp`. The third file, `L.as.ev`, contains prolog code that is only necessary to create an extension of the formalism `L`.

4.2 The AS Centaur-Environment

This section describes the AS-specific part of the Centaur-environment. For general information about using a Centaur editor, please refer to the Centaur Manual.

We suggest, as it is usually done in Centaur, to put the specification of a new language `Foo` in a directory named `.../Foo/syntax`.

The command "centaur-language" can be used as usual. In this case, all the files (but the one named `Foo.rdb`) in the `syntax` directory can be removed.

4.2.1 Compiling a New Specification

The AS-environment pop-up menu contains four entries.

Check Call the type-checker

Compile to LL Compile the specification into Lisp code for creating a Centaur formalism

Compile to Eclipse Compile the specification into prolog code for Typol execution with Eclipse

Write Code Write the generated code into files.

After writing an AS specification, one have to type-check it. If some error occur, the source file must be modified in consequence and re-type-checked until no error occurs. Then you are ready for generating object code. Centaur users must generate code both for Lisp and Prolog. The code that as been generated must then be written down. The generated files will be written in the same directory as the source file.

4.2.2 Getting Help in Case of Errors or Warnings

The AS environment experiments a new way of helping the Centaur user. Each message produced in the error window contains an hyper-link to the HTML form of this documentation.

Assuming that a Netscape WEB browser is running in the X user's environment, clicking on a message with the right button will ask Netscape to show the corresponding part of the manual. The Netscape browser must be running before trying to get help. Its window will pop up to show the manual. Be sure that this window can be visible on your screen.

If necessary, the following resources may be modified.

```
*as.MessageModules.typechecker.*.Help-action:  \
                                #:message-help:netscape
*as.MessageModules.typechecker.*.Help-root: app
*as.MessageModules.typechecker.*.Help-location: \
                  clf/as/doc/as_manual/typechecker.html
```

4.2.3 Using an AS-defined Formalism in the Centaur Environment

When a new formalism has been created using AS, some resources must be updated to make it accessible by Centaur. Some of these resources may have been already created using the "centaur-language" command (See the Centaur documentation).

Assuming a new formalism `Foo`, the .centaur.rdb file must contains the following resources:

```
Centaur.Formalism: (... Foo ...)
?.Foo*Root: user
?.Foo.Database.Location: .../Foo.rdb
```

The main resource file for `Foo` must contain the following resources:

```
?.Foo.Location: Foo/syntax
?.Foo.Mode: std
?.Foo.LoadFunction: #:as:load-a-formalism
```

Note that the last resource is not generated by the "centaur-language" command.

You can either restart a new Centaur, or use the "Reset Resources" then the "Load Formalism" buttons to take your new formalism into account.

4.2.4 Modifying a Specification

When a specification has been modified and the objects files regenerated as explained above, one must either restart a new Centaur or use the "Load Formalism" button. In the last case, this will not reload the formalism into Prolog if Typol has already used this formalism. The simpler way to insure that Prolog get the same version of your formalism as Centaur is to restart a new Prolog server, using the following command in the Lisp top-level window.

```
({prolog}:direct "halt")
```

5 Reference Manual

This section presents the AS language in its entirety. Both syntactical and semantical aspects are covered.

Lexical elements are defined giving their lex definition. We give the concrete syntax of the language using a BNF-like notation with the following conventions:

```
-->     is defined as
|       or
[x]     optional x
*       list
+       non-empty list
```

5.1 Lexical Elements

5.1.1 Keywords

The following words are reserved:

```
abstract end extends integer is of string syntax tree with
```

A keyword can be used as an identifier by prefixing it by a sharp symbol ("#") as in "`#string`".

5.1.2 Identifiers

Identifiers are made of letters (lower or upper case letters), digits, and underscores ("_"). They must begin with a letter and may end by an arbitrary number of quotes ("'").

If a keyword is used as an identifier, it must be prefixed by a sharp symbol ("#").

▷ *Keywords* 5.1.1 *p.*17

The following lex syntax is used to define identifiers.

Syntax:

```
id -->    [a-zA-Z][a-zA-Z0-9\_]*'*
        | #[a-zA-Z][a-zA-Z0-9\_]*'*
```

Upper and lower case letters must be considered as different. An identifier prefixed with a sharp sign is equivalent to one without.

Examples:

```
plus, Exp, #integer, op''
```

5.1.3 Comments

Comments begin with the double minus sign ("–") anywhere on the line and continue to the end of the line.

Example:

```
-- This is a comment
```

5.1.4 Empty Lines

Unlike in most computer languages, empty lines cannot be found anywhere in the text. They can appear only inside a list of declarations and generate a node in the abstract syntax. Such empty lines do not have any semantical meaning, but are there to give a textual structure to the text. This features is only used as a test in the current Centaur environment and may not survive in a future version of the language.

Here is the lex definition of a line:

Syntax:

```
line --> \n[ ]*\n
```

5.2 Names

An abstract syntax is a recursive definition of a formalism, some operators (constructors), and some types (also called phyla) designed by their names. These names are identifiers. A name can be used before it has been define.

There is no rule on the appearance of the identifiers for operators and phyla. One can chose to begin phyla names by an upper case letter, an operator names by a lower case as it is done in most of the examples in this manual, but this rule is not part of the language.

▷ *Identifiers* 5.1.2 *p.*17

The name "meta" is not allowed for an operator as it has a special meaning in the Centaur VTP. When creating a formalism, a special operator named "meta" used to create and manipulate schemes (or incomplete trees) in Centaur is added to every phyla of the syntax.

It is possible to give the same name for an operator and a phylum or a formalism. One can think that this possibility is useful when a phylum contains exactly one operator, for example. However, the AS type-checker will emit a warning when a name is overloaded. Double definitions of an operator or of a phylum are not allowed and result in error messages.

It is not possible to give the same name for a formalism and a phylum of this formalism as the formalism may be viewed as the type union of all the phyla belonging to the formalism.

5.3 Typed Names

When defining a modular definition, one needs to use some phyla belonging to an other formalism. There is no "import" or "use" clause in AS to import names from an other

formalism. One have to use a typed identifier of the form `P::L` where `L` is the name a formalism, and `P` the name of a phylum belonging to this formalism. If "L" is not accessible to the type-checker, or if "P" does not belong to `L`, an error message is produced.

Syntax:

```
tid -->  id
       | id "::" id
```

▷ *Names* 5.2 *p.*18

5.4 Definition of Operators

An operator is defined by giving its signature and its name. Operators with the same signature can be defined at once. The signature of an operator contains its domain (types of its sons) and its codomain.

Operators may be classified into four classes: atomic operators, fixed arity operators, list operators and second-order operators.

5.4.1 Atomic Operators

Atomic operators are operators whose domain are `string`, `integer` or `tree`. They correspond to the terminal of the syntax and carry a value whose type depends on their domain. This class of operators inherited from Metal may be merge with fixed arity operators in a future version of AS.

The following examples give the typical declaration for identifiers or integers:

Examples:

```
id : string -> Id;
int : integer -> Int;
```

5.4.2 Fixed Arity Operators

Fixed arity operators are operators whose domain are a product of some phyla. There is no limit on the size of the product. Operators of arity zero are fixed arity operators (their correspond to the singletons of Metal that was considered as atomic operators).

Examples:

```
pair : Elem # Elem -> Pair;
plus : Exp # Exp -> Exp;
void : -> Option;
```

5.4.3 List Operators

List operators are operators whose domain is of the form "list of `P`" where P is a phylum. As Metal, AS distinguish lists that may be empty ("*-list") from those that cannot be empty ("+-list").

Examples:

```
exp_s : Exp* -> Exp_s;
id_s : Id+ -> Id_s;
```

5.4.4 Second-order Operators

Second-order operators are operators that introduce a binder in the abstract syntax. From a unique declaration, the system generates a first order operator for a first order version of the formalism and a second order operator for a higher-order version of the syntax.

In a λ-calculus like language, the first-order abstract syntax of the expression $\lambda x.E$ is usually defined by `lamb : Id # Exp -> Exp`, as x must be a variable name (an identifier), and E an expression. As $\lambda x.E$ is also a function of type `Exp -> Exp`, its higher-order abstract syntax is usually defined by `lamb : (Exp -> Exp) -> Exp`. In order to allow the use of both the first-order and the second-order view, AS merge these two definitions into a unique one as in the following example. The type-checker verifies that the phylum given for the first-order type of the variable (here `Id`) contains a unique operator whose domain is `string`.

Example:

```
lamb : (Id:Exp -> Exp) -> Exp;
```

5.4.5 Modularity

Notice that the phyla used in the signatures may come from another formalism previously defined and accessible to the system, allowing modular definition of abstract syntaxes.

▷ *Formalism Inclusion* 3.6 *p.*12 - *Typed Names* 5.3 *p.*18

5.4.6 Syntax

We give here the complete syntax of operator declarations:

```
operator  --> id ("," id)* ":" signature ";"
signature --> domain "->" id
domain    -->   "string"
              | "integer"
              | "tree"
              | [tid ("#" tid)*]
              | tid "+"
              | tid "*"
              | "(" tid ":" tid "->" tid ")"
```

5.5 Definition of Phyla

Phyla may be introduced implicitly or explicitly.

An operator definition implicitly defines the codomain of its signature as a phylum if is not already declared. So most of the phyla in an abstract syntax will be implicitly defined.

The definition

```
pair :  Elem # Elem -> Pair;
```

defines `Pair` as a phylum, and `pair` as an operator whose codomain is `Pair`. This definition does not define `Elem` that must be defined somewhere else.

5.5.1 Type Inclusion

A type inclusion implicitly defines the container as a phylum if it is not already declared.

The following example of type inclusion expresses the fact that an identifier is also an expression.

Example:

```
Id < Exp;
```

Predefined types cannot be used in a type inclusion.

Circular definitions of phyla results in equal phyla and will be indicated by the type-checker as a warning.

5.5.2 Union of Phyla

A phylum can also be defined explicitly as the union of other phyla. In this case, it is not possible to include some other phylum in it later on, neither to use it as the co-domain of an operator.

Example:

```
Exp_Stat = Exp + Stat;
```

5.5.3 Modularity

Explicit definition of phyla and phyla inclusion may use some phyla imported from a previously defined formalism, allowing modular definition of abstract syntaxes.

▷ *Formalism Inclusion* 3.6 *p.*12 - *Typed Names* 5.3 *p.*18

5.5.4 Syntax

Here is the complete syntax of phyla inclusion and phyla union:

```
inclusion --> tid "<" sinclusion ";"
sinclusion -->   id
               | id "<" sinclusion
union --> id "=" tid ("+" tid)* ";"
```

5.6 Predefined Types

The following types are predefined and can be used in the domain of atomic operator :

```
integer string tree
```

As they are not phyla, they cannot be modified by any means or be used in the definition of a phyla.

5.7 Complete Definition of an Abstract Syntax

The complete definition of an abstract syntax is given by a name (for the defined formalism) and a list of declaration.

```
program     -->   "abstract" "syntax" "of" id "is"
                     declaration*
                  "end" [id] ";"
declaration -->   operator | inclusion | union | line
```

The last identifier is optional. If present it must be equal to the first one. However, the parser does not check this constraint.

5.8 Extension of a Formalism

The definition of operators or phyla can reuse some phyla previously defined in another formalism.

It is also possible to extend a formalism by adding to a previous definition some new operators and new phyla. The result is a new formalism (there is no reuse in this case) strongly related to the previous one as it is possible to coerce programs from the old formalism into programs in the new one. The reverse can only be done if the operators that have been added are not used.

```
program     -->   "abstract" "syntax" "of" id
                     "extends" id "with" declaration*
                  "end" [id] ";"
```

Only formalisms defined with AS can be extended.

5.9 Effectiveness

Defining an abstract syntax as a type system is not sufficient for practical applications. As the goal of an abstract syntax is to allow to represent programs or structural documents, one will want to be able to construct ground terms belonging to the syntax that is defined. This property is called effectiveness and is required to get the induction principle often used when reasoning and proving properties of programming languages or of programs.

Note that a non-effective syntax can become effective when it is extended. The following syntax is not effective as it does not contains ground terms:

```
abstract syntax of L is

f : X -> X;

end L;
```

We can extend this syntax, adding an atomic operator:

```
abstract syntax of L'
  extends L with

z : -> X;

end L;
```

The resulting abstract syntax is now effective as it contains ground terms.

5.10 The Complete Concrete Syntax of AS

```
program     -->   "abstract" "syntax" "of" id
                     definition
                  "end" [id] ";"
definition  -->   "is" declaration*
                | "extends" id "with" declaration*
declaration -->   operator | inclusion | union | line
operator    -->   id ("," id)* ":" signature ";"
signature   -->   domain "->" id
domain      -->   "string"
                | "integer"
                | "tree"
                | [tid ("#" tid)*]
                | tid "+"
                | tid "*"
                | "(" tid ":" tid "->" tid ")"

inclusion   -->   tid "<" sinclusion ";"
sinclusion  -->   id
                | id "<" sinclusion

union       -->   id "=" tid ("+" tid)* ";"
```

5.11 The Complete Abstract Syntax of AS

Here is the complete definition of AS in AS. Notice that for bootstrapping reasons, the current version of AS is defined in Metal. This explains also why the identifiers used in this definition comply with the Metal rules, i.e. phyla are in upper case and operator in lower case letters.

```
abstract syntax of as is

program : ID # DEFINITION -> PROGRAM;

declaration_s : DECLARATION *      -> DECLARATION_S;
extends       : ID # DECLARATION_S -> DEFINITION;
DECLARATION_S < DEFINITION;

line       :                  -> DECLARATION;
operator   : ID_S # SIGNATURE -> DECLARATION;
phylum     : ID # TID_S       -> DECLARATION;
inclusion  : TID # INCL       -> DECLARATION;
sinclusion : ID # INCL        -> INCLUSION;
ID < INCL;

id  : string  -> ID;
tid : ID # ID -> TID;
ID < TID;

id_s  : ID+  -> ID_S;
tid_s : TID* -> TID_S;

signature : DOMAIN # ID -> SIGNATURE;

#string, #integer, #tree   :                 -> DOMAIN;
star_list_of, plus_list_of : TID             -> DOMAIN;
prod                       : TID*            -> DOMAIN;
lamb                       : TID # TID # TID -> DOMAIN;

end as;
```

6 Errors and Warnings

When using Centaur to type-check or compile an AS specification, some error messages or warnings may appear in a specialized window. Hypertext features are used to help the user to localize and understand these messages.

Some words of the messages are emphasized. By clicking (usually with the left button of the mouse) on one of these words, the user can see in its source window to which piece of the specification this particular word refers. In the following, error messages are presented with variables (X, Y,...) that are replaces by strings or subparts of the program when the message is emitted to form a complete phrase. The verb "refers' when applied to a variable (or to the text that replaces it at run time) designates this hypertext facility.

Each error message produced by the system are commented in the present document. There exists hypertext links between the messages produced by the system and the HTML version of the present documentation. By clicking (usually with the right button of the mouse) on a particular message it is possible to visualize the corresponding documentation (for example using Netscape).

▷ *Getting Help in Case of Errors or Warnings* 4.2.2 *p.*15

6.1 Type-Checker Errors and Warnings

6.1.1 "The name X is already used for Y."

There is a double definition. X refers to a name that as already been defined somewhere else in your specification. Y refers to this previous definition and indicates which kind of definition (formalism, phylum, operator) it is.

This message can be either an error message or a warning. In AS, you can give the same name to define a type (formalism or phylum) and an operator, however in this case a warning message will be emitted by the type checker. Note that it is not possible to give the same name to a formalism and to a phylum belonging to this formalism.

▷ *Names* 5.2 *p.*18

6.1.2 "The type X is not defined."

X refers to the use of an undefined type name. To define a phylum, one have to either define an operator belonging to this phylum or to include some other phylum in it. To refer to a phylum belonging to an other formalism, use the form `P::L` where `P` is the name of a phylum belonging to the formalism `L`.

▷ *Definition of Phyla* 5.5 *p.*21 - *Names* 5.2 *p.*18

6.1.3 "I cannot find the formalism X."

The formalism X cannot be found by the system. X must be a Centaur formalism and must appear in the list given by Centaur when clicking on "Reload/Formalism". Check your resources if this formalism exists and is not accessible by Centaur.

If Centaur is not running on the same computer than Eclipse, verify that both computer can see the same file system.

▷ *Using an AS-defined Formalism in the Centaur Environment* 4.2.3 *p.*15

6.1.4 "The formalism X does not define Y."

X refers to a formalism name. Y refers to a phylum that do not belong to X as expected by the expression Y::X.

▷ *Typed Names* 5.3 *p.*18

6.1.5 "The phylum X is closed."

X refers to a phylum that is closed, i.e. it is not possible to modify this phylum by defining new constructors or by including some other phylum in it.

This is the case for phyla defined as the union of other phyla.

▷ *Union of Phyla* 5.5.2 *p.*21

6.1.6 "The phylum X should be a singleton to be used in a lambda definition."

Only singleton phylum (i.e. containing exactly one constructor) can be used as binder in second-order types.

X refers to a phylum defined locally that contains more than one operator and gives you the list of operators belonging to this phylum.

▷ *Second-order operators* 5.4.4 *p.*20

6.1.7 "The phylum X (= Y) should be a singleton to be used in a lambda definition."

Only singleton phylum (i.e. containing exactly one constructor) can be used as binder in second order types.

X refers to an imported phylum that contains more than one operator. The tag Y gives you the list of operators belonging to this phylum.

▷ *Second-order operators* 5.4.4 *p.*20

6.1.8 "In phylum X, Y should be defined with signature Z to be usable in a lambda definition."

X refers to a phylum name occurrence in a second-order type expression. When X is not an imported phylum, Y refers to the definition of the unique constructor belonging to X. Z gives you the acceptable signature such that Y can be used in a lambda definition.

▷ *Second-order operators* 5.4.4 *p.*20

6.1.9 "The phylum X is not effective."

X refers to a phylum definition. This phylum is not effective, i.e. it is not possible to construct a ground term of type X.

Even if this fact is reported by the type-checker as a warning, most of the time this is an error in the design of your abstract syntax.

However, if the formalism that you are defining is intended to be extended by adding some constructors, X can become effective only after the extension.

▷ *Effectiveness* 5.9 *p.*22

6.1.10 "The name meta is not allowed for an operator."

The operator name "meta" is reserved. When creating a formalism for the VTP of Centaur, a special operator named "meta" used to create schemes (or incomplete trees) in Centaur will be added to every phylum of the syntax. This node must not be defined or redefined in an AS specification. The tag "meta" refers to the incorrect declaration.

▷ *Names* 5.2 *p.*18

6.1.11 "The name X is both defined as Y and imported from Z."

A name is both imported from some external formalism Z as an operator and defined in the present specification, leading to a name conflict. Y refers to the local definition of X and indicate the kind of definition involved, while X and Z refers to the place where it has been imported from Z.

This message can be either an error message or a warning. In AS, you can give the same name to define a type (formalism or phylum) and an operator, however in this case a warning message will be emitted by the type checker. Note that it is not possible to give the same name to a formalism and to a phylum belonging to this formalism.

▷ *Names* 5.2 *p.*18 - *Typed Names* 5.3 *p.*18 - *Complete Definition of an Abstract Syntax* 5.7 *p.*22

6.1.12 "The two phyla X and Y are identical (=Z)."

X and Y refers to two phyla that contains exactly the same constructors. Z gives the list of these constructors.

This may be due to a circular inclusion of phyla.

▷ *Definition of Phyla* 5.5 *p.*21

6.1.13 "The formalism X is not defined using as."

X refers to a formalism that is imported in your specification, but that is not defined using AS. Only formalisms defined with AS can be extended.

▷ *Extension of a Formalism* 5.8 *p.*22

6.1.14 "The file X defining Y cannot be read."

The formalism Y is defined with AS, but for some reason the file necessary to perform its extension cannot be read. It must be in the directory containing its AS specification.

If Centaur is not running on the same computer than Eclipse, check that both computer can see the same file system.

▷ *Using an AS-defined Formalism in the Centaur Environment* 4.2.3 *p.*15

6.1.15 "You can't define and extend X at the same time."

Modular specifications must be stratified. "define" and "extend" refers to the places in your specification where X is used. One of them must be incorrect.

▷ *Formalism Extension* 3.7 *p.*13 - *Complete Definition of an Abstract Syntax* 5.7 *p.*22

6.2 Messages from the AS Centaur-Environment

6.2.1 "The source file has been modified but not saved."

The result of an AS compilation has been written to a file, but the source file was modified and not saved. This inconsistent situation may be the source of future problems.

6.2.2 "Your definition is not yet type-checked."

You asked to compile an AS specification that was not yet type-checked. Type check your specification first, and then compile it.

▷ *Compiling a New Specification* 4.2.1 *p.*14

6.2.3 "The name of the file (X) differs from the name of the language defined (Y)."

The name of an AS file must correspond to the name of the formalism defined in it to allow auto-loading of generated files.

▷ *Files Naming Convention* 4.1 *p.*14

6.2.4 "X written."

The file `X` has been written on your file system.

7 Implementation Notes

The syntax of AS as been specified in Metal and PPML. Its semantics (type-checker and compiler) is written in Typol. The syntax of the environment used by the type-checker is specified in AS. It imports a generic package that implements binary trees in order to get faster access to the environment. The compiler that produces some code for the VTP of Centaur uses the Le_Lisp standard definition of Centaur (LL). The compiler that produces Prolog code uses a definition of a sub-part of Prolog written in AS itself. A compiler generating Prolog/MALI (an implementation of λ-Prolog) has also been written but is no more used.

7.1 Lacks, Bugs and Features

- The injection (coercion) applications between a formalism and its extensions are not yet implemented.
- The type-checker does not report unused phyla.
- It is not yet possible to declare a phylum as non effective to avoid useless warnings.
- When formalism inclusion is used, the type-checker makes the assumption that the syntax of the imported formalism is effective.

7.2 Type-Checking

The AS type-checker has been written in Typol. It produces as result, apart error messages and warnings, two values that are attached as annotations at the root of the type-checked specification. The first one, called `as_env`, contains the complete environment produced by the type-checker and is used as input by the code generators. The second, called `includes`, contains the list of all declarations, included those that are imported in case of the current specification is an extension of a previous one. These two annotations will be saved in a file to allow separate compilation when the current specification must be extended.

The environment contains three distinct parts. All of them contain binary trees. The first one is used for formalisms, the second for phyla, the third for operators. This binary trees are part of a generic package designed for handling environment, with implementations as lists, sorted lists, binary trees and colored binary trees. In the present case, the binary trees implementation gives the best time performances (the overhead used by the colored binary trees kills the advantage of these balanced trees even for large specifications as the C abstract syntax specification).

The environment contains some paths to allow the generation of complete error messages with hyper-text links. These paths are erased in the saved version of the environment as there are of no use in the code generation process.

Two fix points are calculated. The first one checks that the specification is effective (i.e. for each phylum it is possible to construct a ground term), the second calculates the sets of

operators that belong to each phylum, taking type inclusion into account. Most of the time taken by the type-checker is spent when returning (i.e. constructing the effective resulting value) from this last procedure.

The following example will be used in the two next sections to illustrate code generation.

```
abstract syntax of L is

id : string -> Phy;
f : Phy -> Phy;

end L;
```

7.3 Lisp Code Generation

Lisp code generation takes as argument the environment generated by the type-checker. It produces a Lisp function (`#:L:formalism_create` if `L` is the name of the formalism) that creates a VTP formalism. This function is put as an annotation (`LL_code`) to the source specification. It first creates the formalism (`formalism:make`), empty phyla (`phylum:make`), and operators (`operator:make`). It then constructs the effective value of the phyla (`phylum:insert`), creates the "meta" operator then close the formalism (`formalism:complete`).

Here is the Lisp code generated for the previous example:

```
(defun #:L:formalism-create ()
  (lets ((L-form ({formalism}:make 'L))
         (Phy-phy ({phylum}:make 'Phy L-form))
         (f-op ({operator}:make 'f L-form 1 Phy-phy))
         (id-op ({operator}:make 'id L-form 0 {name})))
        ({phylum}:insert Phy-phy f-op)
        ({phylum}:insert Phy-phy id-op)
        ({formalism}:metavarop
            L-form
            ({operator}:make 'meta L-form 0 {metavariable}))
        (addop_meta L-form)
        ({formalism}:complete L-form)
        L-form))
```

Notice that the Centaur VTP allows the use of a phylum belonging to a foreign formalism in the domain of an operator (formalism inclusion). As this was not the case in Mentor, such a formalism cannot be saved in a (`.t`) table. For the same reason, polish (`.po`) files cannot be used with AS defined formalisms.

7.4 Prolog Code Generation

Prolog code generation takes as argument the environment generated by the type-checker and produces Prolog clauses that are put as an annotation (`pl_code`) to the source specification. These clauses feed three requirements of the Typol implementation: a types data-base used by the Typol type-checker, dynamic type-checking and prolog to VTP conversion and dynamic type-checking.

Here is the Prolog code generated for the previous example:

```
%$lang(phylum)
'$L'('Phy').
%$lang(operator,Lang::phylum)
'$L'('f', '$TYPOL$l_id'('$TYPOL$id'('L'),
                        '$TYPOL$id'('Phy'))).
'$L'('id', '$TYPOL$l_id'('$TYPOL$id'('L'),
                         '$TYPOL$id'('Phy'))).
%$lang(operator, sons_type, name)
'$L'(
   'f',
   '$TYPOL$type_s'(
      '$TYPOL$l_id'('$TYPOL$id'('L'), '$TYPOL$id'('Phy')),
      '$TYPOL$type_s'('nil')), '$L$f').
'$L'('id', '$TYPOL$id'('string'), '$L$id').
%$lang$phylum(scheme)
delay '$L$Phy'(_X) if 'var'(_X).
'$L$Phy'('$L$f'(_)).
'$L$Phy'('$L$id'(_)).
%$lang$op(arity,lang,op)
'$L$f'(1, 'L', 'f').
'$L$id'('string', 'L', 'id').
```

7.5 Files Generation

The effective generation of Lisp and Prolog text is done by a PPML pretty-printer. However, the saved environment used to compile formalism extensions is directly written done by Prolog.

References

[1] P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of the 3rd Symp. on Software Development Environments*, Boston, USA, November 1988. Rapport de Recherche INRIA 777, Inria-Sophia-Antipolis, France, December 1987.

[2] R. J. Boulton. Syn: A single language for specifying abstract syntax trees, lexical analysis, parsing and pretty-printing. Technical Report 390, University of Cambridge Computer Laboratory, Mar. 1996.

[3] T. Despeyroux and A. Hirschowitz. Abstract syntax and induction : a theory. Draft, July 1996.

[4] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environment based on structured editors: The mentor experience. Technical report, Inria, July 1980.

[5] V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang, and J.-J. Levy. Programming environment based on structured editors: The mentor experience. In D. Barstow, H. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*. McGraw-Hill, 1984.

[6] J. Hannan. Extended natural semantics. *Journal of Functional Programming*, 3(2):123–152, 1993.

[7] G. Kahn, B. Lang, B. Mélèse, and E. Morcos. Metal: A formalism to specify formalisms. *Science of Computer Programming*, 3(2):151–188, 1983.

[8] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–210, 1993.

[9] B. Lang. The virtual tree processor. In J. Heering, J. Sidi, and A. Verhoog, editors, *Generation of Interactive Programming Environments*. CWI Report, May 1986.

[10] G. Nadathur and D. Miller. An overview of λProlog. In K. A. Bowen and R. A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, Aug. 1988. MIT Press.

[11] G. Nadathur and F. Pfenning. The type system of a higher-order logic programming language. In F. Pfenning, editor, *Types in Logic Programming*, pages 245–283. MIT Press, 1992.

[12] F. Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

Index

abstract syntax, 7, 9
AS Centaur environment, 15
atomic operator, 19

binder, 12, 20
binding, 8

CLF, 7
comment, 17
compiler, 15
Computer Languages Factory, 7

effectiveness, 22
empty line, 18
environment, 13
example, 10
expression, 10

fixed arity operator, 19
formalism, 10, 18
formalism extension, 8, 14
formalism inclusion, 8, 13, 18
formalism loading, 16
formalism reloading, 16
formalism restriction, 14

help, 15
higher-order abstract syntax, 8

identifier, 17
integer, 19

keywords, 17

list, 9, 12
list operator, 19

meta, 18
modularity, 8, 14, 18

name, 18

operator, 7, 9, 10, 18

phylum, 7, 9, 10, 18, 20
predefined type, 21

resource, 16

second-order abstract syntax, 8, 9, 12
second-order operator, 20
sharp, 17
statement, 11
string, 19

tree, 19
type inclusion, 7, 9, 13, 21
type-checker, 15
typed names, 18

union of phyla, 21

Contents

1 Introduction **3**

2 Rational **4**
2.1 The Computer Languages Factory 4
2.2 Abstract Syntax 5
2.3 Isolating Abstract Syntax from Concrete Syntax 5
2.4 Modularity 6
2.5 Second-Order Abstract Syntax 6

3 Tutorial **8**
3.1 Starting with an Example 8
3.2 Expressions 9
3.3 Statements 10
3.4 Lists 11
3.5 Binders 11
3.6 Formalism Inclusion 12
3.7 Formalism Extension 13

4 User's Manual **14**
4.1 Files Naming Convention 14
4.2 The AS Centaur-Environment 14
4.2.1 Compiling a New Specification 14
4.2.2 Getting Help in Case of Errors or Warnings 15
4.2.3 Using an AS-defined Formalism in the Centaur Environment 15
4.2.4 Modifying a Specification 16

5 Reference Manual **17**
5.1 Lexical Elements 17
5.1.1 Keywords 17
5.1.2 Identifiers 17
5.1.3 Comments 18
5.1.4 Empty Lines 18
5.2 Names 18
5.3 Typed Names 18
5.4 Definition of Operators 19
5.4.1 Atomic Operators 19
5.4.2 Fixed Arity Operators 19
5.4.3 List Operators 20
5.4.4 Second-order Operators 20
5.4.5 Modularity 20
5.4.6 Syntax 20

5.5 Definition of Phyla 21
5.5.1 Type Inclusion 21
5.5.2 Union of Phyla 21
5.5.3 Modularity 21
5.5.4 Syntax 21
5.6 Predefined Types 22
5.7 Complete Definition of an Abstract Syntax 22
5.8 Extension of a Formalism 22
5.9 Effectiveness 22
5.10 The Complete Concrete Syntax of AS 23
5.11 The Complete Abstract Syntax of AS 24

6 Errors and Warnings **25**
6.1 Type-Checker Errors and Warnings 25
6.1.1 "The name X is already used for Y." 25
6.1.2 "The type X is not defined." 25
6.1.3 "I cannot find the formalism X." 25
6.1.4 "The formalism X does not define Y." 26
6.1.5 "The phylum X is closed." 26
6.1.6 "The phylum X should be a singleton to be used in a lambda definition." 26
6.1.7 "The phylum X (= Y) should be a singleton to be used in a lambda definition." 26
6.1.8 "In phylum X, Y should be defined with signature Z to be usable in a lambda definition." 26
6.1.9 "The phylum X is not effective." 27
6.1.10 "The name meta is not allowed for an operator." 27
6.1.11 "The name X is both defined as Y and imported from Z." 27
6.1.12 "The two phyla X and Y are identical (=Z)." 27
6.1.13 "The formalism X is not defined using as." 27
6.1.14 "The file X defining Y cannot be read." 28
6.1.15 "You can't define and extend X at the same time." 28
6.2 Messages from the AS Centaur-Environment 28
6.2.1 "The source file has been modified but not saved." 28
6.2.2 "Your definition is not yet type-checked." 28
6.2.3 "The name of the file (X) differs from the name of the language defined (Y)." 28
6.2.4 "X written." 28

7 Implementation Notes **29**
7.1 Lacks, Bugs and Features 29
7.2 Type-Checking 29
7.3 Lisp Code Generation 30
7.4 Prolog Code Generation 31

7.5 Files Generation 31

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399